# Assignment on Decorators

1. **Write a Python program to create a decorator that logs the arguments and return value of a function.**

   **Ans:**

```
def hello_decorator(func):
    def inner1(*args, **kwargs):

        print("before Execution")

        returned_value = func(*args, **kwargs)
        print("after Execution")

        return returned_value

    return inner1


@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b

a, b = 1, 2
print("Sum =", sum_two_numbers(a, b))
```

**OUTPUT:**

```
before Execution
Inside the function
after Execution
Sum = 3
```

2. **Write a Python program to create a decorator function to measure the execution time of a function.**

   **Ans:**

```
import time

def measure_execution_time(func):
    """Decorator to measure the execution time of a function."""

    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function '{func.__name__}' took {execution_time:.4f} seconds to execute.")
        return result
```

```
    return wrapper
@measure_execution_time
def factorial(n):
    """Calculates the factorial of a number."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
factorial(5)
```

**OUTPUT:**

```
Function 'factorial' took 0.0000 seconds to execute.
Function 'factorial' took 0.0000 seconds to execute.
Function 'factorial' took 0.0001 seconds to execute.
Function 'factorial' took 0.0001 seconds to execute.
Function 'factorial' took 0.0001 seconds to execute.
Function 'factorial' took 0.0001 seconds to execute.
```

3. **Write a Python program to create a decorator to convert the return value of a function to a specified data type.**

**Ans:**

```
def convert_to_data_type(data_type):
    def decorator(func):
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            return data_type(result)
        return wrapper
    return decorator

@convert_to_data_type(int)
def add_numbers(x, y):
    return x + y

result = add_numbers(10, 20)
print("Result:", result, type(result))

@convert_to_data_type(str)
def concatenate_strings(x, y):
    return x + y

result = concatenate_strings("Python", " Decorator")
print("Result:", result, type(result))
```

**OUTPUT:**

```
Result: 30 <class 'int'>
Result: Python Decorator <class 'str'>
```

**4. Write a Python program that implements a decorator to cache the result of a function.**

**Ans:**

```
from functools import wraps
# Decorator function to cache the result of a function
def memoize(func):
    cache = {}
    @wraps(func)
    def wrapper(*args, **kwargs):
        key = (args, frozenset(kwargs.items()))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]
    return wrapper
# Example usage
@memoize
def add(x, y):
    print("Calculating...")
    return x + y
result1 = add(3, 4)
result2 = add(3, 4)
print(result1)
print(result2)
```

**OUTPUT:**

```
Calculating...
7
7
```

**5. Write a Python program that implements a decorator to validate function arguments based on a given condition.**

**Ans:**

```
def validate_args(condition):
    """Decorator to validate function arguments based on a condition."""
```

```
    def decorator(func):
        def wrapper(*args, **kwargs):
            if not condition(*args, **kwargs):
                raise ValueError("Invalid arguments")
            return func(*args, **kwargs)

        return wrapper

    return decorator

# Example usage:
@validate_args(lambda x: x > 0)  # Ensure argument is positive
def my_function(x):
    print("Function executed with x = ", x)
my_function(5)

try:
    my_function(-2)
except ValueError as e:
    print("Error:", e)
```

**OUTPUT:**

```
Function executed with x =  5
Error: Invalid arguments
```

6. **Write a Python program that implements a decorator to retry a function multiple times in case of failure.**

**Ans:**

```
 import time
def retry(max_retries=3, delay=1):
    def decorator(func):
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_retries:
                try:
                    result = func(*args, **kwargs)
                    return result
                except Exception as e:
                    print(f"Attempt {attempts + 1} failed with error: {str(e)}")
                    attempts += 1
                    time.sleep(delay)
            raise Exception(f"Function {func.__name__} failed after {max_retries} attempts")

        return wrapper

    return decorator

@retry(max_retries=5, delay=2)
```

```
def example_function():
    # Replace this with the function you want to retry
    import random
    if random.random() < 0.7:
        raise ValueError("Random failure")
    return "Success"

if __name__ == "__main__":
    try:
        result = example_function()
        print(f"Function succeeded: {result}")
    except Exception as e:
        print(f"Function failed: {str(e)}")
```

**OUTPUT:**

```
Attempt 1 failed with error: Random failure
Function succeeded: Success
```

7. Write a Python program that implements a decorator to enforce rate limits on a function.

Ans:

```
import time

def rate_limited(max_calls=5, period=60):
    """Decorator to enforce rate limits on a function."""

    def decorator(func):
        calls = 0
        last_reset = time.time()

        def wrapper(*args, **kwargs):
            nonlocal calls, last_reset

            # Check if rate limit is exceeded
            elapsed = time.time() - last_reset
            if calls >= max_calls and elapsed < period:
                raise Exception("Rate limit exceeded. Try again in {} seconds.".format(int(period -
elapsed)))

            # Reset calls if the period has elapsed
            if elapsed >= period:
                calls = 0
                last_reset = time.time()

            # Increment call count and call the original function
            calls += 1
            return func(*args, **kwargs)
```

```
        return wrapper

    return decorator

# Example usage
@rate_limited(max_calls=3, period=10)  # Allow 3 calls per 10 seconds
def api_call():
    print("Making an API call...")
    # Simulate an API call with a 1-second delay
    time.sleep(1)
    return "API response"

# Call the decorated function multiple times to test rate limiting
for i in range(5):
    try:
        result = api_call()
        print(result)
    except Exception as e:
        print(e)
```

**OUTPUT:**

```
Making an API call...
API response
Making an API call...
API response
Making an API call...
API response
Rate limit exceeded. Try again in 6 seconds.
Rate limit exceeded. Try again in 6 seconds.
```

8. **Write a Python program that implements a decorator to add logging functionality to a function.**

**Ans**:

```
def add_logging(func):
    def wrapper(*args, **kwargs):
        # Log the function name and arguments
        print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")

        # Call the original function
        result = func(*args, **kwargs)

        # Log the return value
        print(f"{func.__name__} returned: {result}")

        # Return the result
```

```
        return result
    return wrapper

# Example usage
@add_logging
def add_numbers(x, y):
    return x + y
result = add_numbers(200, 300)
print("Result:", result)
```

**OUTPUT:**

```
Calling add_numbers with args: (200, 300), kwargs: {}
add_numbers returned: 500
Result: 500
```

**9. Write a Python program that implements a decorator to handle exceptions raised by a function and provide a default response.**

**Ans:**

```
        def exception_handler(default_response):
    def decorator(func):
        def wrapper(*args, **kwargs):
            try:
                result = func(*args, **kwargs)
                return result
            except Exception as e:
                print(f"Exception caught: {e}")
                print(f"Default Response: {default_response}")
                return default_response
        return wrapper
    return decorator

@exception_handler(default_response="An error occurred.")
def example_function():
    result = 1 / 0

example_function()
```

**OUTPUT:**

```
Exception caught: division by zero
Default Response: An error occurred.
[5]:
'An error occurred.'
```

**10. Write a Python program that implements a decorator to enforce type checking on the arguments of a function.**

**Ans:**

```
import inspect

def enforce_type_checking(func):
    """Decorator to enforce type checking on function arguments."""

    def wrapper(*args, **kwargs):
        signature = inspect.signature(func)
        parameters = signature.parameters

        # Check positional arguments
        for i, arg in enumerate(args):
            param_name = list(parameters.keys())[i]
            param_type = parameters[param_name].annotation
            if not isinstance(arg, param_type):
                raise TypeError(f"Argument '{param_name}' must be of type '{param_type.__name__}'")

        # Check keyword arguments
        for param_name, arg in kwargs.items():
            param_type = parameters[param_name].annotation
            if not isinstance(arg, param_type):
                raise TypeError(f"Argument '{param_name}' must be of type '{param_type.__name__}'")

        return func(*args, **kwargs)

    return wrapper
@enforce_type_checking
def add_numbers(x: int, y: float) -> float:
    """Adds two numbers."""
    return x + y

result = add_numbers(5, 3.14)  # No type errors, returns 8.14
print(f"Result: {result}")

try:
    add_numbers("5", 3.14)  # Raises TypeError for invalid type of first argument
except TypeError as e:
    print(e)
```

OUTPUT:

```
Result: 8.14
Argument 'x' must be of type 'int'
```

**11. Write a Python program that implements a decorator to measure the memory usage of a function.**

**Ans:**

```
import tracemalloc

def measure_memory_usage(func):
    """Decorator to measure the memory usage of a function."""

    def wrapper(*args, **kwargs):
        tracemalloc.start()
        result = func(*args, **kwargs)
        snapshot = tracemalloc.take_snapshot()
        top_stats = snapshot.statistics('lineno')

        print(f"Function Name: {func.__name__}")
        print("Top 10 memory-consuming lines:")
        for stat in top_stats[:10]:
            print(f"Line {stat.lineno}: {stat.size / 1024:.2f} KiB")

        tracemalloc.stop()
        return result

    return wrapper

    @measure_memory_usage
def memory_intensive_function():

    large_list = [x**2 for x in range(1000000)]
    return large_list

memory_intensive_function()
```

**OUTPUT:**

```
File "/tmp/sessions/dd96f91aae2346ff/main.py", line 23
    def memory_intensive_function():
IndentationError: unexpected unindent
```

12. Write a Python program that implements a decorator to provide caching with expiration time for a function.

Ans:

```
import functools
import time

def cache_with_expiry(expiry_time=60):
    """Decorator to cache function results with expiration time."""

    def decorator(func):
        cache = {}
```

```
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            key = (args, tuple(kwargs.items()))
            if key in cache and time.time() - cache[key][1] < expiry_time:
                return cache[key][0]


            result = func(*args, **kwargs)
            cache[key] = (result, time.time())
            return result

        return wrapper

    return decorator
@cache_with_expiry(expiry_time=30)
def calculate_multiply(x, y):
    """Calculates the product of two numbers."""
    time.sleep(2)
    return x * y

result1 = calculate_multiply(4, 5)
print(result1)

result2 = calculate_multiply(4, 5)
print(result2)
time.sleep(35)
result3 = calculate_multiply(4, 5)
print(result3)
```

**OUTPUT:**

```
20
20
20
```

# Function

**1. Write a function that inputs a number and prints the multiplication table of that number.**

**Ans:**

```
def print_multiplication_table(n):
    """Prints the multiplication table of a given number up to 12."""
    for i in range(1, 13):
        print(f"{n} x {i} = {n*i}")
print_multiplication_table(7)
```

**OUTPUT:**

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
>
```

**2. Write a program to print twin primes less than 1000. If two consecutive odd numbers are both prime then they are known as twin primes.**

**Ans:**

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True
def find_twin_primes(limit):
    twin_primes = []
    for num in range(3, limit, 2):
        if is_prime(num) and is_prime(num + 2):
            twin_primes.append((num, num + 2))
    return twin_primes
limit = 1000
```

```
result = find_twin_primes(limit)

print("Twin Primes less than", limit, "are:")

for twin_prime in result:

    print(twin_prime[0], "and", twin_prime[1])
```

**OUTPUT:**

```
Twin Primes less than 1000 are:
3 and 5
5 and 7
11 and 13
17 and 19
29 and 31
41 and 43
59 and 61
71 and 73
101 and 103
107 and 109
137 and 139
149 and 151
179 and 181
191 and 193
197 and 199
227 and 229
239 and 241
269 and 271
281 and 283
311 and 313
347 and 349
419 and 421
431 and 433
461 and 463
521 and 523
569 and 571
599 and 601
617 and 619
641 and 643
659 and 661
809 and 811
821 and 823
827 and 829
857 and 859
881 and 883
```

**3. Write a program to find out the prime factors of a number. Example: prime factors of 56 - 2, 2, 2, 7.**

**Ans**:

```
def prime_factors(num):
    factors = []
    divisor = 2

    while divisor <= num:
        if num % divisor == 0:
            factors.append(divisor)
            num //= divisor
        else:
            divisor += 1

    return factors

# Example: Find prime factors of 56
number_to_factorize = 56
result = prime_factors(number_to_factorize)
print(f"Prime factors of {number_to_factorize} are: {result}")
```

**OUTPUT:**

```
Prime factors of 56 are: [2, 2, 2, 7]
```

**4. Write a program to implement these formulae of permutations and combinations.**

**Ans:**

```
def sum_of_divisors(num):
    divisor_sum = 1  # Start with 1 as every number is divisible by 1
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            divisor_sum += i
            if i != num // i:
                divisor_sum += num // i
    return divisor_sum
def are_amicable(num1, num2):
    return sum_of_divisors(num1) == num2 and sum_of_divisors(num2) == num1
# Example usage:
number1 = 220
number2 = 284
if are_amicable(number1, number2):
    print(f"{number1} and {number2} are amicable numbers.")
else:
    print(f"{number1} and {number2} are not amicable numbers.")
```

**OUTPUT:**

```
220 and 284 are amicable numbers.
```

**5. Write a function that converts a decimal number to binary number.**

**Ans:**

```
def decimal_to_binary(n):
  """
  Converts a decimal number to its binary representation.

  Args:
    n: The decimal number to convert.

  Returns:
    A string representing the binary equivalent of the input number.
  """
```

```
  if n == 0:
    return "0"
  binary_string = ""
  while n > 0:
    remainder = n % 2
    binary_string = str(remainder) + binary_string
    n //= 2
  return binary_string

# Example usage
decimal_number = 10
binary_representation = decimal_to_binary(decimal_number)
print(f"{decimal_number} in binary is: {binary_representation}")
```

**OUTPUT:**

```
10 in binary is: 1010
```

6. **Write a function cubesum() that accepts an integer and returns the sum of the cubes of individual digits of that number. Use this function to make functions PrintArmstrong() and isArmstrong() to print Armstrong numbers and to find whether is an Armstrong number.**

**Ans:**

```
def cubesum(number):
    digit_cubes = [int(digit) ** 3 for digit in str(number)]
    return sum(digit_cubes)

def isArmstrong(number):
    return number == cubesum(number)

def PrintArmstrong(limit):
    armstrong_numbers = [num for num in range(limit + 1) if isArmstrong(num)]
    print(f"Armstrong numbers up to {limit}:")
    for armstrong_number in armstrong_numbers:
        print(armstrong_number)
limit = 500
PrintArmstrong(limit)
```

**OUTPUT:**

```
Armstrong numbers up to 500:
0
1
153
370
371
407
```

7. **Write a function prodDigits() that inputs a number and returns the product of digits of that number.**

**Ans:**

```
def prodDigit(number):
    # Initialize product to 1
```

```
    product = 1

    # Convert the number to a string to iterate through its digits
    str_number = str(number)
    # Iterate through each digit and multiply it with the product
    for digit in str_number:
        # Convert digit back to integer before multiplying
        product *= int(digit)

    return product

# Example usage:
input_number = int(input("Enter a number: "))
result = prodDigit(input_number)
print("Product of digits:", result)
```

**OUTPUT:**

```
Enter a number: 1234
Product of digits: 24
```

8. **If all digits of a number n are multiplied by each other repeating with the product, the one digit number obtained at last is called the multiplicative digital root of n. The number of times digits need to be multiplied to reach one digit is called the multiplicative persistance of n.**

**Ans:**

```
def multiply_digits(number):
    result = 1
    for digit in str(number):
        result *= int(digit)
    return result

def multiplicative_digital_root(number):
    while number >= 10:
        number = multiply_digits(number)
    return number

def multiplicative_persistence(number):
    persistence_count = 0
    while number >= 10:
        number = multiply_digits(number)
        persistence_count += 1
    return persistence_count

# Example usage:
input_number = int(input("Enter a number: "))
digital_root = multiplicative_digital_root(input_number)
persistence = multiplicative_persistence(input_number)
```

```
print(f"Multiplicative Digital Root: {digital_root}")
print(f"Multiplicative Persistence: {persistence}")
```

**OUTPUT:**

```
Enter a number: 367
Multiplicative Digital Root: 2
Multiplicative Persistence: 3
>
```

9. **Write a function sumPdivisors() that finds the sum of proper divisors of a number. Proper divisors of a number are those numbers by which the number is divisible, except the number itself. For example proper divisors of 36 are 1, 2, 3, 4, 6, 9, 18 .**

**Ans:**

```
def sumPdivisors(n):
    if n <= 1:
        return 0

    divisors_sum = 1  # Start with 1 as 1 is a proper divisor for all numbers
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:  # Avoid counting the same divisor twice for perfect squares
                divisors_sum += n // i

    return divisors_sum

# Example usage:
number = 36
result = sumPdivisors(number)
print(f"The sum of proper divisors of {number} is: {result}")
```

**OUTPUT:**

```
The sum of proper divisors of 36 is: 55
```

10. **A number is called perfect if the sum of proper divisors of that number is equal to the number. For example 28 is perfect number, since 1+2+4+7+14=28. Write a program to print all the perfect numbers in a given range.**

**Ans:**

```
def is_perfect(n):
 """
 Checks if a number is perfect.

 Args:
   n: The number to be checked.

 Returns:
   True if the number is perfect, False otherwise.
```

```
    """
    if n <= 1:
      return False
    sum_of_divisors = 1
    for i in range(2, int(n**0.5) + 1):
      if n % i == 0:
        sum_of_divisors += i + n // i
    return sum_of_divisors == n

def print_perfect_numbers(lower, upper):
    """
    Prints all perfect numbers in a given range.

    Args:
      lower: The lower bound of the range (inclusive).
      upper: The upper bound of the range (inclusive).
    """
    for num in range(lower, upper + 1):
      if is_perfect(num):
        print(num, end=" ")
# Example usage
lower_bound = 2
upper_bound = 1000
print(f"Perfect numbers between {lower_bound} and {upper_bound}:")
print_perfect_numbers(lower_bound, upper_bound)
```

**OUTPUT:**

```
Perfect numbers between 2 and 1000:
> 5
6 28 496 5
```

**11.** Two different numbers are called amicable numbers if the sum of the proper divisors of each is equal to the other number. For example 220 and 284 are amicable numbers.

Ans:

```
def sum_of_divisors(num):
    divisor_sum = 1  # Start with 1 as every number is divisible by 1
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            divisor_sum += i
            if i != num // i:
                divisor_sum += num // i
    return divisor_sum
def are_amicable(num1, num2):
    return sum_of_divisors(num1) == num2 and sum_of_divisors(num2) == num1
```

```
# Example usage:
number1 = 220
number2 = 284
if are_amicable(number1, number2):
    print(f"{number1} and {number2} are amicable numbers.")
else:
    print(f"{number1} and {number2} are not amicable numbers.")
```

**OUTPUT:**

```
220 and 284 are amicable numbers.
```

**12. Write a program which can filter odd numbers in a list by using filter function.**

**Ans:**

```
a = [2, 5, 7, 8, 10, 13, 16]
result = filter(lambda x: x % 2 == 1, a)
print('Original List :', a)
print('Filtered List :', list(result))
```

OUTPUT:

```
Original List : [2, 5, 7, 8, 10, 13, 16]
Filtered List : [5, 7, 13]
>
```

**13. Write a program which can map() to make a list whose elements are cube of elements in a given list.**

**Ans:**

```
def cube(x):
    return x ** 3
input_list = [1, 2, 3, 4, 5]
cubes_list = list(map(cube, input_list))
print("Original List:", input_list)
print("Cubes List:", cubes_list)
```

**OUTPUT:**

```
Original List: [1, 2, 3, 4, 5]
Cubes List: [1, 8, 27, 64, 125]
>
```

**14. Write a program which can map() and filter() to make a list whose elements are cube of even number in a given list.**

**Ans:**

```
def cube_of_even(numbers):
    # Use filter() to select even numbers
    even_numbers = filter(lambda x: x % 2 == 0, numbers)

    # Use map() to calculate the cube of each even number
    cubes_of_even = map(lambda x: x ** 3, even_numbers)

    # Convert the result to a list
```

```
    result_list = list(cubes_of_even)

    return result_list

print("Original List:", input_list)
print("Cubes of Even Numbers:", output_list)
```

**OUTPUT:**

```
>>>input_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>output_list = cube_of_even(input_list)

Original List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Cubes of Even Numbers: [8, 64, 216, 512, 1000]
```